TUTORIAL

BY MAI ELBAABAA

On Your Seats (1/4)

Write Correct / Incorrect, Tight / Not Tight

1. $4n^2 - 300n + 12 \in O(n^2)$	CORRECT, TIGHT
2. $4n^2 - 300n + 12 \in O(n^3)$	CORRECT, NOT TIGHT
3. $3^n + 5n^2 - 3n \in O(n^2)$	INCORRECT, NOT TIGHT
4. $3^n + 5n^2 - 3n \in O(4^n)$	CORRECT, NOT TIGHT
5. $3^n + 5n^2 - 3n \in O(3^n)$	CORRECT, TIGHT
6. $50 \bullet 2^n n^2 + 5n - \log(n) \in O(2^n)$	INCORRECT, NOT TIGHT

Exercise 2

Write True or False :

$$T(n) = 5n^3 + 2n^2 + 4 \log n$$

1. $T(n) \in O(n^4)$
2. $T(n) \in O(n^2)$
3. $T(n) \in \Theta(n^3)$
4. $T(n) \in O(\log n)$
5. $T(n) \in \Theta(n^4)$
6. $T(n) \in \Omega(n^2)$

- Rules of thumb
- in describing the asymptotic complexity of an algorithm:
  - If the running time is the sum of multiple terms, keep the one with the largest growth rate and drop the others, since they will not have an impact for large
  - If the remaining term is a product, drop any multiplicative constants

$$f(n) \approx 1000 + n + 2n^2 \longmapsto f(n) \approx n^2 \qquad f(n) \approx 2^n + 2^{n+1} \longmapsto f(n) \approx 2^{n+1}$$

$$f(n) \approx 2 + 400n + 2^n \longmapsto f(n) \approx 2^n \qquad f(n) \approx n + 500 \log n \longmapsto f(n) \approx n$$

$$f(n) \approx 2\sqrt{n} + 500 \log n \longmapsto f(n) \approx \sqrt{n} \qquad f(n) \approx \sqrt{n} + n + n^3 \longmapsto f(n) \approx n^3$$

MATH BACKGROUND: EXPONENTS

- Some useful identities:
  - $X^A \cdot X^B = X^{A+B}$
  - $X^A / X^B = X^{A-B}$
  - $(X^A)^B = X^{AB}$
  - $X^N + X^N = 2X^N$
  - $2^N + 2^N = 2^{N+1}$

MATH BACKGROUND: LOGARITHMS

- Logarithms
  - *definition*: $X^A = B$ if and only if $\log_X B = A$
  - *intuition*: $\log_X B$ means:
    "the power X must be raised to, to get B"
  - In this course, a logarithm with no base implies base 2.
    $\log B$ means $\log_2 B$
- Examples
  - $\log_2 16 = 4$ (because $2^4 = 16$)
  - $\log_{10} 1000 = 3$ (because $10^3 = 1000$)

- **O(1):** Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion, and call to any other non-constant time function.

- **O(n):** Time Complexity of a loop is considered as O(n) if the loop variables are incremented/decremented by a constant amount. For example following functions have O(n) time complexity.

```
// Here c is a positive integer constant
 for (int i = 1; i <= n; i += c) {
     // some O(1) expressions
 }
```

- **O(n^c):** Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example, the following sample loops have O(n^2) time complexity

```
for (int i = 1; i <=n; i += c) {
     for (int j = 1; j <=n; j += c) {
        // some O(1) expressions
     }
 }

 for (int i = n; i > 0; i -= c) {
     for (int j = i+1; j <=n; j += c) {
        // some O(1) expressions
 }
```

- **O(Logn)** Time Complexity of a loop is considered as O(Logn) if the loop variables are divided/multiplied by a constant amount.

```
for (int i = 1; i <=n; i *= c) {
      // some O(1) expressions
  }
  for (int i = n; i > 0; i /= c) {
      // some O(1) expressions
  }
```

- What is the exact runtime and complexity class (Big-Oh)?

```
int sum = 0;
for (int i = 1; i <= N; i += c) {
    sum++;
}
```

  - Runtime = N / c = O(N).

```
int sum = 0;
for (int i = 1; i <= N; i *= c) {
    sum++;
}
```

  - Runtime = logc N = O(log N).

Call this number of multiplications "x".

$2^x = N$

$\boldsymbol{x = \log_2 N}$

- After getting the above problems. Let's have two iterators in which, outer one runs **N/2 times,** and we know that the time complexity of a loop is considered as ***O(log N)***, if the iterator is divided / multiplied by a constant amount **K** then the time complexity is considered as $\boldsymbol{O(\log_K N)}$.
- *$(N/2)^K = 1$ (for k iterations)*
  => $N = 2^k$ (taking log on both sides)
  => $k = \log(N)$ base 2.
  Therefore, the time complexity will be
  $T(N) = O(\log N)$

EXAMPLE: $O(N^2)$

```java
public static void main(String[] args) {
    int n = 100;
    for (int i =1 ; i <= n/3; i++) {
        for (int j = 1; j < n; j=j+4) {
            System.out.println("*");
            break;
        }
    }
}
```

outer loop will run n/3 times
inner loop will run n/4 times
so total time complexity is $(n/3)*(n/4)=n^2/12=O(n^2)$

- *What is the exact runtime complexity (Big-Oh)?*

```
int sum = 0;
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N * 2; j++) {
        sum++;
    }
}
```

  - *Runtime = N · 2N = O(N^2).*

```
int sum = 0;
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= i; j++) {
        sum++;
    }
}
```

  - *Runtime = N (N + 1) / 2 = O(N^2).*

- **Arithmetic series:** $\sum_{k=1}^{n} k = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$

```
if (value % 2 == 0){
    return true;
  }
  else
    return false;
}
```

Answer: O(1). Constant run time complexity.

Because you're only ever taking one value, there is no "loop" to go through.

```
for (let i=0; i<array.length; i++) {
        if (array[i] === item) {
            return i;
        }
    }
```

Answer: O(n). Linear run time complexity.

HOW TO FIND COMPLEXITY?

Some rules of thumb

Basically just count the number of statements executed:

- If there are only a small number of simple statements in a program — **O(1)**
- If there is a 'for' loop dictated by a loop index that goes up to *n* — **O(*n*)**
- If there is a nested 'for' loop with outer one controlled by *n* and the inner one controlled by *m* — **O(*n**m*)**
- For a loop with a range of values *n*, and each iteration reduces the range by a fixed constant fraction (eg: ½) — **O(log *n*)**